Fraser International College

CMPT 135 202003 Final Project

Due Date and Time: Saturday 12 December 2020 at 8:00AM PST

Vision of Project

In this project we are going to build a software solution for the computation of the shortest path from one place to another place in a two dimensional space similar to a global positioning system (GPS) without making use of any C++ library; instead building everything from scratch.

The vision of the project is that given the map of a region and its road connectivity, we would like to compute the shortest path to go from one place to another place. See the diagram below for illustration purposes.

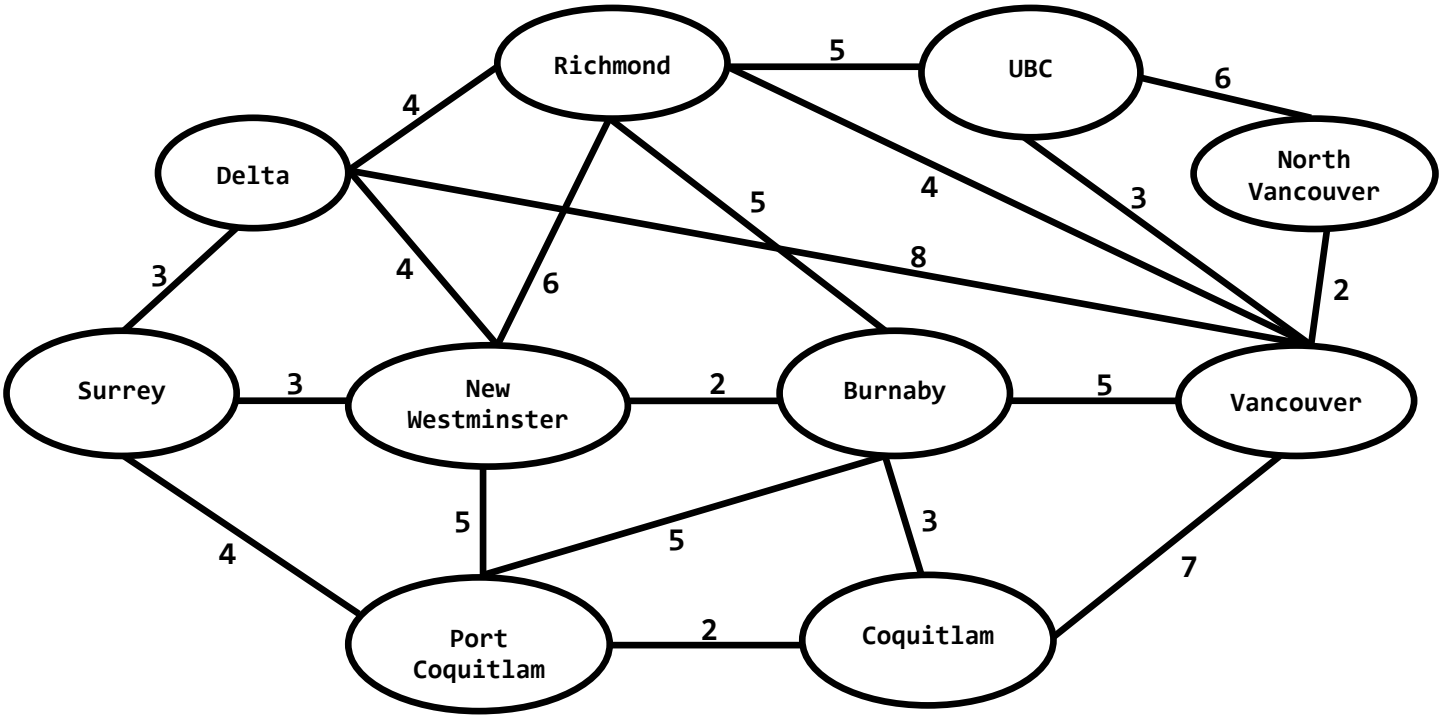

Fig 1: Region map and its road connectivity

As we can see in the illustration above, places in the region are connected by roads. There may or may not be a direct road from a place to another. In the illustration above for example, there is a direct road from Vancouver to North Vancouver but there is no a direct road from Vancouver to Port Coquitlam. However, we can go from Vancouver to Port Coquitlam in several ways among which are the following paths (or routes)

Vancouver → Burnaby → Port Coquitlam

Vancouver → Coquitlam → Port Coquitlam

Vancouver → Burnaby → New Westminster → Port Coquitlam

Moreover, we assume that the roads are two ways and that whenever there is a road from A to B then we automatically assume that there is a road from B to A. That is, we assume that roads are symmetric by nature.

Thus we may go from Vancouver to Port Coquitlam using the following routes as well

Vancouver → Delta → Surrey → Port Coquitlam

Vancouver → Coquitlam → Burnaby → Port Coquitlam

Or if we want to waste our time going in an unnecessary loop, then we may go in the following route too

Vancouver →Burnaby →Richmond → New Westminster → Burnaby → Port Coquitlam

We can still find many more possible routes.

The aim of our project is therefore be to find the shortest path. Of course in order to compute the shortest path, we need to be given the distances of the roads as shown on the illustration above.

In computing science, we generally speak in terms of **cost** of a path. The cost can be any measurable quantity as you go from one place to another place. Typical costs include distances between places (like the illustration shown above), or the amount of petrol you will burn as you drive from one place to another, or some other measurements.

For example if the cost under consideration is the amount of petrol you burn as you go from one place to another, then it should also be noted that you may burn more petrol on a given shorter distance than another longer distance. For example if the given shorter road is uphill but the longer road is flat. This is how a typical GPS device (application) works.

Therefore, we need to always define what measurement to use to compute costs of paths so that we can select the shortest path; that is the path with a minimum cost.

In the above illustration for example, assuming the cost is the distances shown then the costs of the six paths we have listed will respectively be: **10**, **9**, **12**, **15**, **14**, and **22**. Since the minimum cost among the costs computed is 9, we therefore conclude that the shortest path among the paths shown above (which happens to be the shortest path among all possible paths) is therefore to go through the path

Vancouver → Coquitlam → Port Coquitlam [cost = 9]

Of course it should be noted from the paths listed above that the last one is a useless computation because the path contains a loop (that is to say the path passes through a place but then loops back to the same place before it reaches the destination place). A path with a loop should obviously be avoided.

Terminologies

- **Graph:** The region map with its road connectivity shown in the diagram above is called a graph.
- **Vertex:** A vertex (plural vertices) is a unit in a graph (such as a city in the diagram above).
- **Edge:** A unit in a graph that connects pairs of vertices of the graph.
- **Path:** A route to go from one vertex of a graph to another following the edges in the graph.
- **Origin Vertex of an edge:** A vertex from where an edge emanates.
- **Destination Vertex of an edge:** A vertex where an edge ends.

Scope of Project

The scope of the project will encompass

- **Data Structures:** The design of classes to represent vertices, edges, graphs, and paths in a graph.
- **Algorithms:** The design of clearly defined steps to solve the shortest path problem.

Restrictions of Libraries

We would like to build the project from scratch. Thus we will limit ourselves to using only a few libraries of C++ language; namely

```
#include <iostream>
#include <fstream>,
#include <cassert>
#include <ctime>
#include <iomanip>
```

No other library is permitted in the project.

Starting Our Project

At this stage, it should be fairly obvious that our vertices have names that need to be stored in our application. For this reason, it makes sense to start by defining a class for a string of characters. The following class declaration based on Assignment 1 therefore suffices to achieve our objective.

```
class CMPT135_String
{
private:
    char *buffer;
public:
    CMPT135_String(); //The buffer is initialized to nullptr value
    CMPT135_String(const char *); //A null terminated char array
    CMPT135_String(const CMPT135_String &); //Deep copy constructor
    ~CMPT135_String();  //Delete any heap memory and assign buffer nullptr value
    CMPT135_String& operator = (const CMPT135_String &); //Deep copy assignment
    int length() const; //Return the number of printable characters
    bool empty() const;  //Return true if length is 0. Otherwise return false
    char& operator [] (const int &) const;  //Assert index and then return the char at the index
    CMPT135_String operator + (const char &) const; //See assignment 1
    CMPT135_String& operator += (const char &); //See assignment 1
    bool operator == (const CMPT135_String &) const; //See assignment 1
    bool operator != (const CMPT135_String &) const; //See assignment 1
    friend istream& operator >> (istream &, CMPT135_String &); //Implemented for you
    friend ostream& operator << (ostream &, const CMPT135_String &); //Implemented for you
};
istream& operator >> (istream &in, CMPT135_String &s)
{
    //This function reads from keyboard or file characters until either a TAB, EOL, or EOF is reached
    //The function ignores any leading or trailing spaces.

    //Define some useful constant values
    #define SPACE ' '
    #define TAB '\t'
    #define EOL '\n'

    //Delete the old value of s
    s.~CMPT135_String();
```

```
    //Skip leading spaces, tabs, and empty lines
    char ch;
    while (!in.eof())
    {
        in.get(ch);
        if (ch == SPACE || ch == TAB || ch == EOL)
            continue;
        break;
    }

    //Append the value in ch to s
    if (ch != SPACE && ch != TAB && ch != EOL)
        s += ch;

    //Read characters into s until a TAB or EOL or EOF is reached
    while (!in.eof())
    {
        in.get(ch);
        if (ch == TAB || ch == EOL || in.eof())
            break;
        else
            s += ch;
    }

    //Remove any trailing spaces
    int trailingSpacesCount = 0;
    for (int i = s.length()-1; i >= 0; i--)
    {
        if (s[i] != SPACE)
            break;
        trailingSpacesCount++;
    }
    CMPT135_String temp;
    for (int i = 0; i < s.length()-trailingSpacesCount; i++)
        temp += s[i];
    s = temp;

    return in;
}
ostream& operator << (ostream &out, const CMPT135_String &s)
{
    for (int i = 0; i < s.length(); i++)
        out << s[i];
    return out;
}
```

The input/output streaming operator functions are implemented for you in order to avoid unnecessary too much time on technicalities.

Next, we will need to store several items (such as vertices, edges, CMPT135_Strings) in different containers. It therefore makes sense to define a generic container using a class template so that we may also use the same container to store different data types. The following class declaration based on Assignment 3 therefore suffices to achieve our objective.

```
template <class T>
class SmarterArray
{
private:
    T *A;
    int size;

public:
```

```
    //Constructors
    SmarterArray(); //Implemented for you
    SmarterArray(const T*, const int&); //Deep copy of the argument
    SmarterArray(const SmarterArray<T>&); //Deep copy of the argument

    //Assignment operator
    SmarterArray<T>& operator = (const SmarterArray<T>&); //Memory clean up and deep copy of the argument

    //Destructor
    ~SmarterArray(); //Memory clean up

    //Getters, Setters, operators and other functions
    int getSize() const; //Return the number of elements in the container
    T& operator[](const int&) const; //Assert index and then return the element at the given index
    int find(const T&) const; //Return the index of the first element that is == to the argument.
                              //Return -1 if not found.
    void append(const T&);  //Store the argument value after the last element
    void insert(const int &, const T&); //Assert the integer argument index >= 0 && index <= size and then
                                        //Insert the T type argument into the calling object at the index.
                                        //If the integer argument is equal to size, then perform append
    bool remove(const int&); //If the index is valid, then remove the element at the index argument
                            //from the calling object and return true. Otherwise return false.
                            //You don't need to assert the index argument.
    bool operator == (const SmarterArray<T>&) const; //return true if sizes are equal and
                                                     //elements at same indexes are equal

    template <class T1>
    friend ostream& operator << (ostream&, const SmarterArray<T1>&); //Implemented for you
};
template <class T>
SmarterArray<T>::SmarterArray()
{
    this->A = nullptr;
    this->size = 0;
}
template <class T>
ostream& operator << (ostream& out, const SmarterArray<T>& L)
{
    if (L.getSize() == 0)
        out << "[Empty List]";
    else
    {
        for (int i = 0; i < L.getSize()-1; i++)
            out << L[i] << endl;
        out << L[L.getSize()-1] << endl;
    }
    return out;
}
```

The output stream operator friend function is implemented for you so that everyone gets same output format.

Task 1

Implement the **`CMPT135_String`** class and the **`SmarterArray`** class that we will be using in our subsequent sections.

Representing vertices, edges, and a graph

Each Vertex object will store the name of the city (**CMPT135_String** data type) at the vertex and a set of edges emanating (coming out) from the vertex together with their costs. A Graph will then simply be a set of vertices (that is a **SmarterArray<Vertex>** data type). This means each Vertex object will have an associated index in the graph. Therefore an Edge object can easily be represented as a struct that will store the index of its destination vertex in the underlying graph and the cost of the edge. Thus we are given the following C++ struct declaration to represent Edge objects.

```
struct Edge
{
    int desVertexIndex; //the index (in the graph) of the destination vertex of this edge
    double cost; //cost of an edge
};
```

We note that an Edge object does not need to store its origin vertex because the origin vertex will store the edge instead.

The following class declaration therefore suffices in order to represent vertices of a graph.

```
class Vertex
{
private:
    CMPT135_String name; //name of the vertex
    SmarterArray<Edge> E; //edges emanating from this vertex. All the elements of E have the same origin
//vertex which is the this object. But they have different destination vertices which are given by the
//desVertexIndex member variable of each element

public:
    Vertex(); //Assign name = "N/A" and initialize E to an empty container
    Vertex(const CMPT135_String &); //Assign name = the argument and initialize E to an empty container
    CMPT135_String getName() const; //Return the name
    SmarterArray<Edge> getEdgeSet() const; //Return E
    int getEdgeSetSize() const; //Return the size of E
    Edge getEdge(const int &) const; //Assert an edge whose destination vertex index is equal to the
                                     //argument exists in E and then return it
    double getEdgeCost(const int &desVertexIndex) const; //Assert an edge whose destination vertex index
                                     //is equal to the argument exists in E and then return its cost
    void appendEdge(const int &desVertexIndex, const double &cost); //Assert there is no existing edge
              //whose destination vertex index and cost are equal to the argument values and then
              //append a new element whose destination vertex index and cost are initialized with the
              //argument values to E
    friend ostream& operator << (ostream &, const Vertex &); //Implemented for you
};
ostream& operator << (ostream &out, const Vertex &vertex)
{
    out << "Name = " << vertex.name << endl;
    out << "Edge Set" << endl;
    for (int i = 0; i < vertex.E.getSize(); i++)
        out << "\t to ---> " << vertex.E[i].desVertexIndex << ", cost = " << vertex.E[i].cost << endl;
    return out;
}
```

Using this class to represent vertices of a graph, we can see that the city of **Surrey** vertex in the diagram above is represented with a Vertex object whose name member variable is assigned "Surrey" and whose E member variable will store 3 Edge data type elements (**e1 = to the Delta vertex with a cost of 3**, **e2 = to the New Westminster vertex with a cost of 3**, and **e3 = to the Port Coquitlam vertex with a cost of 4**). Assuming the Delta, New Westminster, and Port Coquitlam Vertex objects will be stored at indices 5, 2, and 9 respectively in the graph, then **E[0] = [desVertexIndex = 5, cost = 3]**, **E[1] = [desVertexIndex = 2, cost = 3]**, and **E[2] = [desVertexIndex = 9, cost = 4]**.

Now, we give the class declaration of the Graph class that represents graph objects.

```
class Graph
{
private:
    SmarterArray<Vertex> V;
public:
    Graph();//Construct empty graph
    Graph(const char *); //Construct a graph from a text file whose path is given by the argument
        //The text file input will consist in every line a pair of cities and the cost to go from one to
        //the other. The pair of cities and their cost will be separated by one or more SPACE or TAB
        //characters. It doesn't matter how many spaces or tabs are present. BUT THERE HAS TO BE AT LEAST
        //ONE TAB CHARACTER between the pairs of cities and AT LEAST ONE TAB between the second city and
        //the cost. This is because the CMPT135_String class uses TAB as a deliminator (separator). This
        //means city names can be made up of one or more words separated by spaces. An example of one line
        //of text in the input text file is:
        //        New Westminster            Port Coquitlam       4.5
        //In this example, there has to be at least one TAB char between "New Westminster" and "Port
        //Coquitlam" and at least one TAB character between "Port Coquitlam" and 4. Of course there can be
        //more than one TAB characters. The number of TABs can be equal or different. There are can be
        //zero or more spaces as much as you like without causing any problem. Moreover there can be as
        //many empty lines as one likes at the end of the file. However there MUST NOT BE any empty line
        //before the last line consisting of city pair and cost.
        /*
        Thus this function should perform the following tasks
              1.  Construct a non-default file input streaming object using the argument file name
              2.  Assert the file is opened successfully
              3.  While EOF is not reached
                     a.  Read city name (CMPT135_String data type). This is the departure city.
                     b.  If departure city is empty CMPT135_String object, then break.
                     c.  Read city name (CMPT135_String data type). This is the destination city.
                     d.  Read the cost
                     e.  Append a new vertex whose name is the departure city and whose edge set is empty.
                         You must use the appendVertex member function to append appropriately.
                     f.  Append a new vertex whose name is the destination city and whose edge set is
                         empty. You must use the appendVertex member function to append appropriately.
                     g.  Append a new edge from the departure city to the destination city with a cost
                         read in part (d) above.
                     h.  Append a new edge from the destination city to the departure city with a cost
                         read in part (d) above.
              4.  Close the input file stream object and you are done.
        */
    SmarterArray<Vertex> getVertexSet() const; //Return V
    int getVertexSetSize() const; //Return the number of elements of V
    Vertex getVertex(const int &) const; //Assert the index argument and then return the element at index
    int getVertexIndex(const CMPT135_String &) const; //Return the index of an element whose name matches
                                                       //the argument. If no such element is found, return -1
                                                       //Assertion is not required
    int getVertexIndex(const Vertex &) const; //Return the index of the element whose name matches the
                                              //name of the vertex argument. If no such element is found,
                                              //return -1. Assertion is not required
    CMPT135_String getRandomVertexName() const; //Pick a vertex at random and return its name
    void appendVertex(const Vertex &); //Append the argument only if no such vertex already exists
                                      //If same name vertex already exists then do nothing (just return)
                                      //Assertion is not required
    void appendVertex(const CMPT135_String &); //Append a new vertex with the given name and empty E
    void appendEdge(const CMPT135_String &dep, const CMPT135_String &des, const double &cost); //Assert
//two vertices whose names match the arguments exist. Then append an edge to the vertex whose name matches
//the dep argument. The destination vertex index of the edge must be set to the index of the vertex whose
//name matches des and its cost must be set to the cost argument
    friend ostream& operator << (ostream &, const Graph &); //Implemented for you
};
ostream& operator << (ostream &out, const Graph &g)
{
    const int CITY_NAME_WIDTH = 25;
```

```
    out << endl;
    out << "The graph has " << g.getVertexSetSize() << " vertices." << endl;
    out << "These vertices are" << endl;
    for (int i = 0; i < g.getVertexSetSize(); i++)
    {
        Vertex v = g.V[i];
        out << "Vertex at index " << i << " = " << v.getName() << endl;
    }
    out << endl;
    out << "Each vertex together with its edge set looks like as follows" << endl;
    for (int i = 0; i < g.getVertexSetSize(); i++)
    {
        Vertex v = g.V[i];
        out << v << endl;
    }
    out << endl;
    out << "The graph connectivities are as follows..." << endl;
    out.setf(ios::fixed | ios::left);    //Left aligned fixed decimal places formatting
    for (int i = 0; i < g.getVertexSetSize(); i++)
    {
        Vertex depVertex = g.getVertex(i);
        SmarterArray<Edge> E = depVertex.getEdgeSet();
        for (int j = 0; j < E.getSize(); j++)
        {
            int desVertexIndex = E[j].desVertexIndex;
            Vertex desVertex = g.getVertex(desVertexIndex);
            out << depVertex.getName() << setw(CITY_NAME_WIDTH - depVertex.getName().length()) << " ";
            out << desVertex.getName() << setw(CITY_NAME_WIDTH - desVertex.getName().length()) << " ";
            out << setprecision(2) << E[j].cost << endl;
        }
    }
    out.unsetf(ios::fixed | ios::left); //Removing formatting
    cout.precision(0);              //Resetting the decimal places to default
    return out;
}
```

Observation

- Considering our graph constructor is designed to insert edges not only from departing city to destination city but also vice versa, we conclude that it is assumed in our application that whenever there is a connectivity from a city to another city then there is the same connectivity in the opposite direction. That is to say connectivities are assumed symmetric by nature.
- Thus when we prepare the input text file, we don't need to list symmetric connectivities in the input file; that is it is enough to list a path from "A" to "B" with some cost because our graph constructor member function will add both the edges from "A" to "B" and from "B" to "A" with the same cost.
- Therefore given a text file as follows (See the empty lines at the end of the file? But don't worry they will not cause any problem because the design of our CMPT135_String class and the design of the non-default constructor of our Graph class will take care of it seamlessly).

Connectivity Map.txt

A	**B**	**1.5**
A	**C**	**2.9**
B	**C**	**3.6**
B	**D**	**4.7**
C	**D**	**5.8**

The following test program should produce the outputs shown below.

```
int main()
{
    srand(time(0));
    Graph g("Connectivity Map.txt");
    cout << "Graph constructed successfully." << endl;
    cout << g << endl;
    system("Pause");
    return 0;
}
```

The outputs will be as follows.

```
Graph constructed successfully.

The graph has 4 vertices.
These vertices are
Vertex at index 0 = A
Vertex at index 1 = B
Vertex at index 2 = C
Vertex at index 3 = D

Each vertex together with its edge set looks like as follows
Name = A
Edge Set
         to ---> 1, cost = 1.5
         to ---> 2, cost = 2.9

Name = B
Edge Set
         to ---> 0, cost = 1.5
         to ---> 2, cost = 3.6
         to ---> 3, cost = 4.7

Name = C
Edge Set
         to ---> 0, cost = 2.9
         to ---> 1, cost = 3.6
         to ---> 3, cost = 5.8

Name = D
Edge Set
         to ---> 1, cost = 4.7
         to ---> 2, cost = 5.8


The graph connectivities are as follows...
A                       B                       1.50
A                       C                       2.90
B                       A                       1.50
B                       C                       3.60
B                       D                       4.70
C                       A                       2.90
C                       B                       3.60
C                       D                       5.80
D                       B                       4.70
D                       C                       5.80

Press any key to continue . . .
```

Task 2

Implement the **`Vertex`** class and the **`Graph`** class that we will be using in our subsequent sections.

Representing a path in a graph

A path in a graph will be represented as a series of vertices. In order to conserve memory however, we will not store the actual Vertex objects along the path. Instead we will store only the names of the vertices. In order to find the costs of the edges along the path and the total cost of the path (which is the sum of the costs of the edges along the path) then, we will need the underlying Graph object. Moreover the underlying Graph object will help us to validate a given path is a valid path in the graph.

Thus the following class declaration suffices to represent Path objects.

```
class Path
{
private:
    SmarterArray<CMPT135_String> p; //The names of the vertices along the path
public:
    Path(); //Construct an empty path
    int length() const; //Return the number of vertices in the path (the number of elements of p)
    int find(const CMPT135_String &) const; //Return the index of element of p whose name matches the
                                            //argument. If no element satisfies the condition, then return -1
    double computePathCost(const Graph &) const; //Compute the sum of the costs of edges along this path
        //given the underlying graph argument. Remember that the path object stores only city names. Thus
        //you need the underlying graph argument to determine the vertices in the graph that belong to the
        //cities. Then you will be able to find the edges that connect the vertices which will enable you to
        //get the costs of the edges. The sume of the costs of these edges is returned from this function.
    CMPT135_String& operator [] (const int &) const; //Assert index is valid and then return the
                                            //element of p at the given index
    void append(const CMPT135_String &); //Append the argument to the calling object
    void insert(const int &index, const CMPT135_String &); //Assert the condition index >= 0 &&
                                            //index <= the length and then insert the CMPT135_String argument
                                            //at the specified index
    void remove(const int &); //Assert the index argument and then remove the element at the specified index
    friend ostream& operator << (ostream &, const Path &); //Implemented for you.
};
ostream& operator << (ostream &out, const Path &p)
{
    out << "[";
    if (p.length() > 0)
    {
        for (int i = 0; i < p.length()-1; i++)
            out << p[i] << " -> ";
        out << p[p.length()-1];
    }
    out << "]";
    return out;
}
```

Computing All Possible Paths: Pseudo-Code

In order to compute all the possible paths, all we need is to utilize our understanding of recursion as described below. Suppose we would like to go from a given city named **departure** to another city named **destination** in an underlying graph named **g**. Then the following algorithm implemented as a non-member function will do the job.

Path computeMinCostPath(const Graph &g, const CMPT135_String &departure, const CMPT135_String &destination, Path &currentPath = Path())

The function will compute and print all the possible paths and then return the one with the minimum cost. Because, we will need the information of the current partial path as we walk through the edges in the underlying graph, this current partial path needs to pass as an argument to the function. Moreover, when we

first call this function from the main program, we need to start from an empty partial path which is why it is initialized to a default empty Path object argument value. This way, we don't need to pass this argument when we call the function from within our main program.

Detailed specification of the shortest path algorithm:

1. Assert there is at least one vertex in the graph g.
2. Compute depVertexIndex = index of the vertex in g whose name matches departure city
3. Compute desVertexIndex = index of the vertex in g whose name matches destination city
4. Assert both depVertexIndex and desVertexIndex are valid
5. If departure == destination, then
    - Congratulations… A path is found.
    - Define a new Path object named minCostPath and assign it the currentPath
    - Append the destination city name to the minCostPath object
    - Print the minCostPath
    - Return the minCostPath
6. Else if the departure city name already exists in the current path, then
    - Too bad… This is becoming a loop that we have to avoid it
    - Define a new Path object named minCostPath and assign it the currentPath
    - Return the minCostPath
7. Else we proceed as follows
8. Find the vertex object in g whose name matches departure city (call this depVertex)
9. Compute the edge set of depVertex object (call this E)
10. Define a new Path object named **minCostPath** (default empty object)
11. Append the name of depVertex to currentPath
12. For i = 0; i < E.getSize(); i++ do the following
    - Find the name of the destination vertex of the edge E[i] in g. Call this nextVertexName
    - If nextVertexName is found in currentPath then we don't need to walk along E[i] because it will create a loop. So we skip this and go to the next iteration our for loop
    - Compute a path starting from nextVertexName to deparature passing also the current path as an argument. We can do this by calling the same function that we are implementing (recursion). The function call line of code is given to you for clarity purposes

      ```
      Path candidatePath = computeMinCostPath(g, nextVertexName, destination, currentPath);
      ```

    - If candidatePath is empty then it is useless, so we skip it. We go to the next iteration in our for loop.
    - If the last element of the candidatePath is not equal to the departure city name, then it means the candidatePath did not reach the destination city. It means it got stuck somewhere and returned with a partial path. Thus it is useless and hence we discard it and we go to the next iteration in our for loop.
    - If the minCostPath object (declared in step 10 above) is empty, then it means the candidatePath is our first successful path and hence we should assign it to the minCostPath. So assign minCostPath = candidatePath and go to the next iteration in the for loop.
    - If the candidatePath has a lower cost than the minCostPath, then update the minCostPath by assigning minCostPath = candidatePath and go to the next iteration in the for loop.
13. Remove the last element of the currentPath. (N.B:This statement is outside the for loop)
14. Algorithm (function implementation) complete!!!

Backtracking Algorithms

The most important tasks in the algorithm above are the **`appending of the depVertex to the current path`** (Step 11) and the **`removal of the last element of current path`** (Step 13). That is in order to go say from departure place A to destination place D and assuming A has edges to B and C then you first walk to B and try to reach along that route to D. Although you will then walk from B to other possible places through valid edges in order to reach D, the fact that you will perform the walking in a recursive way means that when you finish the route through B; you will still remember you came to B from A. At that point, you walk back (i.e. you backtrack) to A and then restart searching for another route along the C.

This is the fundamental idea behind what are known as **`backtracking algorithms`** that allow us to solve complicated problems by doing something and then backtracking in order to try to find all possible solutions to a problem and then pick the best solution.

Backtracking algorithms are used in solving variety of problems such as

- Finding all the permutations of characters in a string,
- Finding all the subsets of a set,
- Finding all the solutions of Sudoku game board,
- Finding all possible moves in a chess game,
- Find the best scheduling times of exams or class times in a school,
- Finding the least number of colors needed to color objects stacked together in some form so that no adjacent objects are colored with the same color, and so on so forth.

Task 3

Implement the **`Path`** class and the **`computeMinCostPath`** non-member function. This completes the project.

Testing Your Work

You may use the following test program to test your work.

```
int main()
{
    srand(time(0));
    Graph g("Connectivity Map.txt");
    cout << "Graph constructed successfully." << endl;
    cout << g << endl;

    CMPT135_String departure = g.getRandomVertexName();
    CMPT135_String destination = g.getRandomVertexName();
    cout << "Computing shortest path from " << departure << " to " << destination << endl;
    Path minCostPath = computeMinCostPath(g, departure, destination);
    cout << endl;
    cout << "Departure: " << departure << endl;
    cout << "Destination: " << destination << endl;
    if (minCostPath.length() == 0)
        cout << "No path found." << endl;
    else
        cout << "The minimum cost path is: " << minCostPath << " with cost = " <<
minCostPath.computePathCost(g) << endl;

    system("Pause");
    return 0;
}
```

Sample Run Output

Assuming the Connectivity Map.txt file is given as follows:

Connectivity Map.txt

A	**B**	**1.5**
A	**C**	**2.9**
B	**C**	**3.6**
B	**D**	**4.7**
C	**D**	**5.8**

Then below is a possible sample run output.

```
Graph constructed successfully.

The graph has 4 vertices.
These vertices are
Vertex at index 0 = A
Vertex at index 1 = B
Vertex at index 2 = C
Vertex at index 3 = D

Each vertex together with its edge set looks like as follows
Name = A
Edge Set
         to ---> 1, cost = 1.5
         to ---> 2, cost = 2.9

Name = B
Edge Set
         to ---> 0, cost = 1.5
         to ---> 2, cost = 3.6
         to ---> 3, cost = 4.7

Name = C
Edge Set
         to ---> 0, cost = 2.9
         to ---> 1, cost = 3.6
         to ---> 3, cost = 5.8

Name = D
Edge Set
         to ---> 1, cost = 4.7
         to ---> 2, cost = 5.8


The graph connectivities are as follows...
A                       B                       1.50
A                       C                       2.90
B                       A                       1.50
B                       C                       3.60
B                       D                       4.70
C                       A                       2.90
C                       B                       3.60
C                       D                       5.80
D                       B                       4.70
D                       C                       5.80

Computing shortest path from A to D
```

```
Path found: [A -> B -> C -> D] with cost 10.9
Path found: [A -> B -> D] with cost 6.2
Path found: [A -> C -> B -> D] with cost 11.2
Path found: [A -> C -> D] with cost 8.7

Departure: A
Destination: D
The minimum cost path is: [A -> B -> D] with cost = 6.2
Press any key to continue . . .
```

Of course this is a very small input file designed to help you test your work minimally. A bigger input file that contains the graph diagram given above is posted for you together with this assignment. A sample run output is also provided for your reference.

Starter Code

You are given a text file that contains all the code segments given in the discussions above. You are required to copy the given code and provide all the missing parts and get the test program to work without any syntax, linking, runtime, or semantic errors. Please note that

- You are NOT allowed to change or modify any function signature. Use them exactly as they are given.
- You are NOT allowed to add any include directive or namespace.
- You are NOT allowed to add any member, friend, or non-member function. You must follow the specifications described in the discussion above and get the starter code to work.
- You are not allowed to remove any member, friend, or non-member function.

Submission

You are required to submit one source code.cpp file that contains all the class declarations, definitions, friend functions implementations, non-member function implementations and the test program. Your code must be able to run smoothly without any syntax, linking, runtime, or semantic errors except for the possible adjustment I need to make to the input file path specification. This means you may save your input file anywhere in your computer and you don't have to worry I will not find your path in my computer. I will do the required adjustment when I mark your submission.

Afterwards…

Once you are done with your project and submitted it, then go ahead and construct graphs from your countries places and use your GPS application to move around your country.

Have fun and enjoy your last touch of CMPT 135 course at FIC!!!